Parallel Acceleration of Circuit simulation Using OpenMP and CUDA

Project: High Performance Optimization of Circuit simulation
Author: Guanghao Xu
Semester: Fall 2025

Abstract

In this project, I investigate parallel acceleration solutions for a computationally intensive scientific simulation program. The original implementation is a serial CPU-based code, which exhibits limited performance when processing large-scale input data.

To improve performance, two parallelization approaches are explored:

1. shared-memory parallelism using **OpenMP**, and
2. GPU acceleration using **CUDA**.

I redesign key computational kernels to exploit multi-core CPU parallelism and massive GPU parallelism, respectively. Experimental results demonstrate that both approaches achieve significant speedup over the serial baseline, with CUDA delivering the highest performance improvement. The project also analyzes the impact of execution parameters such as thread count and CUDA block size on overall performance.

---

1. Introduction

Scientific simulations and large-scale numerical computations often suffer from long execution times when implemented using purely serial algorithms. As problem sizes increase, such implementations fail to fully utilize modern hardware resources, resulting in limited performance and scalability.

The program studied in this project performs iterative numerical computations over large data structures and was originally implemented as a serial CPU-based code. Although functionally correct, its single-threaded execution model and loop-intensive structure constrain performance on contemporary multi-core CPUs and GPUs. With the availability of shared-memory parallelism and GPU acceleration, parallel programming models such as OpenMP and CUDA provide potential opportunities to improve computational efficiency.

---

2. Project Goals:

- To identify the dominant computational kernels in the original code
- To parallelize the code using OpenMP on multi-core CPUs
- To further accelerate computation using CUDA on NVIDIA GPUs
- To evaluate performance improvements and analyze scalability

---

3. Experimental Environment

All experiments were conducted on a Linux-based system equipped with a high-performance CPU, sufficient system memory, and a modern NVIDIA GPU. The experimental platform is summarized as follows:

- **CPU**: Intel® Core™ i9-14900KF (24 physical cores, 32 logical CPUs, 3.2 GHz base frequency, up to 6.0 GHz turbo)
- **System Memory**: 62 GB RAM (2.0 GB swap, not actively used)
- **GPU**: NVIDIA GeForce RTX 4090 (24 GB global memory)
- **Operating System**: 64-bit Linux (single NUMA node)
- **GPU Driver**: NVIDIA driver version 535.183.01
- **CUDA Runtime**: CUDA 12.2

For GPU acceleration, all CUDA experiments were executed on the RTX 4090 in default compute mode. The available system memory capacity was sufficient to hold all required data structures, ensuring that performance measurements were not affected by memory paging or swapping effects.

During development on Windows, **Visual Studio 2022 Community Edition** was used as the primary development environment, with **CUDA Toolkit 13.1** installed. CUDA code was compiled targeting the **sm_75** architecture to ensure compatibility and performance evaluation across different GPU platforms.

This hardware and software configuration provides adequate CPU parallelism, memory bandwidth, and GPU computational capability to evaluate the performance impact of OpenMP-based multi-threading and CUDA-based GPU acceleration.

---

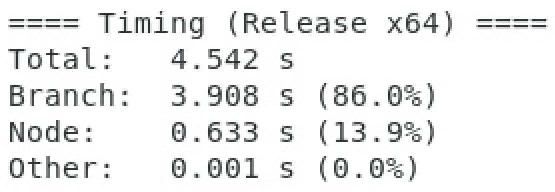
4. Implementation

4.1 Baseline Serial Implementation

The original program is a CPU-based serial implementation written in C++. It performs iterative numerical computations over large data structures, where each iteration updates arrays representing the simulation states of nodes and branches.

In its original form, the program executes in a single-threaded manner and relies heavily on large nested loops to traverse all nodes and branches at each time step. These loops involve frequent memory accesses and do not employ explicit vectorization or parallel execution, resulting in limited utilization of modern multi-core hardware architectures.

Consequently, the execution time increases rapidly as the problem size grows. Profiling results indicate that the majority of the runtime is dominated by repeated loop-based computations, which exhibit a high degree of data parallelism. This computational pattern makes the program a strong candidate for acceleration using parallel programming models such as OpenMP and CUDA.



```
==== Timing (Release x64) ====
Total:   4.542 s
Branch:  3.908 s (86.0%)
Node:    0.633 s (13.9%)
Other:   0.001 s (0.0%)
```

4.2 OpenMP Acceleration

4.2.1 Parallelization Strategy

OpenMP is used to parallelize the most time-consuming loops in the program. These loops exhibit minimal data dependency across iterations, making them suitable for shared-memory parallel execution. However, the computational cost per iteration may vary due to conditional logic and non-uniform memory access behavior.

The following parallelization techniques are applied:

- ***#pragma omp parallel for*** is used to enable loop-level parallelism
- Dynamic scheduling is employed to mitigate potential load imbalance across threads

- Independent memory access patterns are preserved across iterations to avoid false sharing

4.2.2 OpenMP Configuration

The OpenMP version parallelizes the major loop computations using #pragma omp parallel for directives. In the implementation, a dynamic scheduling policy is explicitly applied to distribute loop iterations among threads:

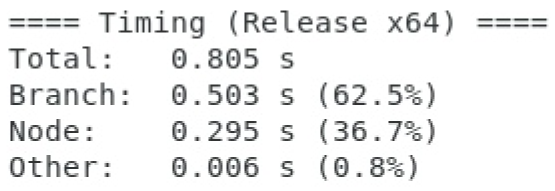
- **schedule(dynamic, 128)** is used for branch loop region.
- **schedule(dynamic, 256)** is used in node loop region

The use of dynamic scheduling allows threads to request new chunks of iterations as they finish previous work, which is suitable when the per-iteration cost may vary across indices (e.g., due to conditional logic or non-uniform memory behavior). The specified chunk sizes (128 and 256) control the granularity of work assignment: each thread receives a block of 128 or 256 consecutive iterations per scheduling request, balancing load distribution and scheduling overhead.

The OpenMP thread count is controlled via the OMP_NUM_THREADS environment variable during experiments.

4.2.3 Analysis

OpenMP provides a relatively low-effort approach to parallelization and achieves noticeable speedup on multi-core CPUs. However, scalability is limited by the number of available CPU cores and shared-memory bandwidth. The choice of scheduling policy and chunk size can significantly affect thread work distribution efficiency and overall runtime performance. As a result, different hardware platforms may require tuning of these parameters to achieve optimal performance.



```
==== Timing (Release x64) ====
Total:   0.805 s
Branch:  0.503 s (62.5%)
Node:    0.295 s (36.7%)
Other:   0.006 s (0.8%)
```

4.3 CUDA Acceleration

While OpenMP improves performance on CPUs, GPUs offer significantly higher parallelism and memory bandwidth. CUDA was therefore employed to further accelerate the computational kernels.

4.3.1 Kernel Design

The primary loop-based computations in the program are refactored into CUDA kernels to exploit GPU parallelism. In the CUDA implementation, each GPU thread is responsible for processing a single computational element, such as a node or a branch, enabling a large number of threads to execute concurrently. This thread-level mapping allows the program to leverage the massive parallel execution capability of modern GPUs and efficiently handle large-scale data-parallel workloads.

Explicit memory management is employed to support GPU execution. Device memory is allocated to store the required data structures, and data is transferred between host and device memory prior to and after kernel execution. Synchronization mechanisms are used to ensure correct execution order and data consistency between CPU and GPU computations.

4.3.2 CUDA Configuration

The CUDA implementation uses the following configuration parameters:

- **Block size**: 64 threads per block
- **Grid size**: computed based on the total number of computational elements
- **Execution model**: single CUDA stream

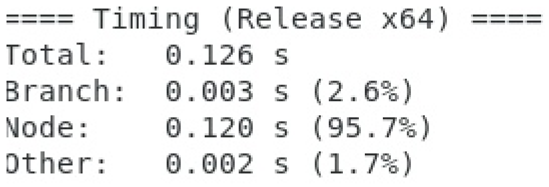
The block size of 64 threads per block is selected to balance occupancy, register usage, and scheduling overhead on the target GPU architecture. A single-stream execution model is adopted because kernel execution dominates the runtime, and using a single stream reduces the number of synchronization points between kernel launches and memory operations, thereby minimizing synchronization overhead and overall execution time. In this workload, overlapping data transfers with computation provides limited additional benefit.

To improve memory efficiency, global memory accesses are coalesced where possible. Shared memory is not utilized, as the computation exhibits limited data reuse across threads, making shared memory unlikely to offer significant performance improvements in this case.

4.3.3 Analysis

CUDA-based acceleration provides substantially higher parallelism compared to the OpenMP implementation by exploiting the massive number of GPU threads and high memory bandwidth. The data-parallel structure of the computation maps naturally to the GPU execution model, enabling efficient utilization of thread-level parallelism for large problem sizes.

However, GPU performance remains sensitive to kernel configuration and memory access patterns. Parameters such as block size influence occupancy and scheduling efficiency, while global memory behavior can significantly affect overall runtime. In addition, the benefits of advanced optimization techniques, such as shared memory usage or multi-stream execution, are workload dependent. For the current application, limited data reuse and kernel-dominated execution reduce the potential advantages of these techniques, making a simpler configuration sufficient. As with OpenMP, parameter tuning on different GPU architectures may be required to achieve optimal performance.



```
==== Timing (Release x64) ====
Total:   0.126 s
Branch:  0.003 s (2.6%)
Node:    0.120 s (95.7%)
Other:   0.002 s (1.7%)
```

---

5. Performance Results

Version	**Execution Time (s)**	**Speedup**
Serial	**4.542s**	**0x**
OpenMP	**0.805s**	**5.64x**
CUDA	**0.126s**	**36x**

---

6. scalability

6.1 OpenMP solution

The scalability of the accelerated implementation depends on both hardware limitations and runtime scheduling efficiency. For OpenMP, performance scaling is constrained by the number of available CPU cores and shared-memory bandwidth, as well as the impact of loop iteration scheduling on thread-level workload distribution. Different scheduling strategies and chunk sizes can alter task allocation overhead and load balancing behavior, which may result in different optimal configurations across platforms.

For the OpenMP implementation, the primary tuning directions include **(1) chunk size selection** and **(2) scheduling strategy selection**. The goal is to choose appropriate chunk sizes that balance per-thread workload and parallelism while reducing scheduling overhead. In addition, because the branch loop updates *cknod[n1].xum* and *cknod[n2].xum* from multiple threads, contention on shared node entries may limit scalability. An alternative approach is to use **thread-local accumulation buffers** (or a two-stage reduction) for *xum* updates, which can reduce synchronization or atomic pressure and improve scalability at higher core counts.

6.2 CUDA solution

The current executable is based on **skylim.cu**, a partially tuned CUDA implementation derived from **skylim.cpp**. A baseline version, **skylim_normal_cuda.cpp**, provides a straightforward CUDA parallelization and serves as a reference point for evaluating optimization impact. Current optimizations focus on kernel parallelism and memory access, with further tuning guided by profiling.

- **Performance profiling with Nsight Systems and Nsight Compute**

  Nsight Systems is used to analyze end-to-end execution, exposing kernel launch frequency, synchronization points, memory transfers, and GPU idle gaps. Nsight Compute complements this view by revealing kernel-level details such as occupancy, memory bandwidth utilization, warp stall reasons, and register pressure. Together, these tools enable correlation between high-level runtime behavior and low-level hardware efficiency, guiding targeted optimization.

- **Reducing synchronization overhead and enabling overlap**

  Frequent host–device synchronization limits scalability when performed at every simulation step. Synchronization can be reduced by batching monitored data transfers or deferring transfers until output is required. When intermediate results are needed, CUDA streams and events can be used to overlap computation with asynchronous memory transfers, improving overall GPU utilization without changing algorithmic correctness.

- **Block and grid configuration tuning**

  Kernel performance is sensitive to launch configuration, and a single fixed block size is unlikely to be optimal. Sweeping block sizes (e.g., 64–512 threads) and limited thread coarsening helps balance occupancy, register usage, and memory throughput. Profiling-driven parameter sweeps are used to identify configurations that maximize SM utilization while avoiding excessive register pressure.

- **Memory access optimization and data layout**

Pinned host memory can improve transfer throughput when used with asynchronous or batched transfers, but offers limited benefit if synchronization remains on the critical path. On the device side, restructuring frequently accessed fields from an AoS to a partial SoA layout can improve memory coalescing. The use of __restrict__ pointers helps the compiler reduce aliasing assumptions and generate more efficient memory access code.

- **Mitigating atomic contention in branch kernels**

Atomic updates to shared node accumulators can become a major bottleneck when many branches map to the same nodes. Contention can be reduced through block-level aggregation or multi-stage gather–scatter approaches that limit global atomic operations. These techniques trade implementation complexity for improved scalability and higher effective throughput.

- **Reducing divergence and instruction overhead**

Conditional logic within kernels may cause warp divergence, especially for special-case nodes or branches. Separating rare cases into dedicated kernels reduces divergence for the common path. Small helper functions should be inlined where appropriate, and loop unrolling can be applied selectively to reduce loop overhead when supported by profiling evidence.

- **Kernel launch reduction and fusion**

When kernel launch overhead becomes significant, reducing the number of launches per simulation step can improve performance. Kernel fusion or batching multiple operations into a single launch may be beneficial if data dependencies allow. These approaches reduce launch and synchronization overhead at the cost of reduced modularity.

- **Compilation and numerical trade-offs**

All builds should use optimized release configurations. Optional techniques such as aggressive compiler optimizations or reduced-precision arithmetic (e.g., FP16 or fixed-point) may offer additional speedups but require careful validation of numerical accuracy. Any performance–accuracy trade-offs must be clearly documented.

7. Conclusion

The experimental results show that both OpenMP and CUDA significantly improve performance compared to the serial baseline, confirming that the workload exhibits strong data parallelism. The OpenMP implementation achieves a speedup of approximately **5.64×**, effectively utilizing multi-core CPU resources, but its scalability is limited by the number of available CPU cores, shared-memory bandwidth, and scheduling overhead from dynamic task allocation.

In contrast, the CUDA implementation achieves a much higher speedup of approximately **36×**, primarily due to massive thread-level parallelism and high memory bandwidth on modern GPUs. The data-parallel structure of the circuit simulation maps naturally to the GPU execution model, enabling efficient concurrent execution. GPU performance remains sensitive to kernel configuration and memory access behavior; in this project, a block size of 64 threads per block and a single-stream execution model provide a balanced trade-off between performance and implementation complexity, as limited data reuse reduces the benefit of shared memory and multi-stream optimization.

Overall, OpenMP offers a practical and low-cost approach for accelerating existing CPU-based code, while CUDA provides substantially higher performance for large-scale data-parallel workloads when increased implementation complexity is acceptable.

NOTE: Development notes are provided in the project root directory to facilitate future maintenance and extension. As the Linux and Windows implementations differ slightly in their library dependencies, users are advised to obtain the Linux-specific code when building or running the project on a Linux platform.